STARS
REUSABILITY GUIDELINES

April 30, 1990

Contract No. F19628-88-D-0032
Task IR40: Repository Integration

Delivered as part of:
CDRL Sequence No. 1550

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscomb AFB, MA 01731-5000

Prepared by:

IBM Systems Integration Division
800 North Frederick
Gaithersburg, MD 20879

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 30, 1990	Final

4. TITLE AND SUBTITLE

~~Repository Guide~~ See Below

5. FUNDING NUMBERS

C: F19628-88-D-0032

6. AUTHOR(S)

R. Ekman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

IBM Federal Sector Division
800 N. Frederick Avenue
Gaithersburg, MD 20879

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CDRL Sequence No. 1550

11. SUPPLEMENTARY NOTES

Software Technology for Adaptable Reliable Systems

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A guide to software reuse using the STARS Repository. This document contains the IBM STARS Repository Guidebook, STARS Repository User's Guide, and STARS Reusability Guidelines. Each is described below.

IBM STARS Repository Guidebook: A guide to the STARS Repository, providing high-level information for all users -- component reusers, component suppliers, and repository administrators. The Guidebook is organized according to the specific roles that users perform when using the system.

STARS Repository User's Guide: A guide on how to access and use the STARS Repository. It provides the basic information needed to use the repository software, but it is not a comprehensive guide to the VAX computer, on which the repository is built.

STARS Reusability Guidelines. A set of Ada coding guidelines for component development that emphasize reusability. Code that follows these guidelines will be easier to reuse on multiple projects and platforms. Many examples are provided illustrating the guidelines.

14. SUBJECT TERMS	15. NUMBER OF PAGES
STARS, software reuse, software reuse library, Ada coding guidelines, Ada	201
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Abstract

This document describes the STARS project Ada coding guidelines, as they relate to reusability. This document is part of a suite of documents that define the use of the IBM STARS Repository. The other documents are the *IBM STARS Repository GuideBook* and the *IBM STARS Repository User's Guide.*

Preface

The guidelines in this document apply to all IBM STARS team tasks. In particular, they apply to Ada code and related documentation as it is admitted to and managed within the IBM STARS Repository.

These guidelines were originally published in the *Consolidated Reusability Guidelines* [IBM0380]. Essentially, the guidelines are the same as the guidelines which were collectively established by the STARS prime contractors in the STARS Q-Increment. Some of the original guidelines were modified for clarity and depth of definition. A few were eliminated due to experiences and comments since they were published. The guidelines have also been adjusted to match the metrics produced by the Repository metric collection tool.

This document was developed by the IBM Systems Integration Division, located at 800 North Frederick, Gaithersburg, MD 20879. Questions or comments should be directed to the author, Robert W. Ekman at (301) 240-6431, or to the IBM STARS Program Office.

Table of Contents

Introduction

This document presents a set of Ada coding guidelines for component development that emphasize reusability. By following these guidelines, your code will be easier to reuse across multiple projects and platforms. Compliance to these guidelines will increase the level of acceptance within the Repository component evaluation process.

Where appropriate, the following information is provided:

- A brief statement of the guideline,
- A detailed explanation of its meaning,
- A rationale on why the guideline is needed,
- Suggestions on how to apply the guideline, and
- Examples to illustrate the use of the guideline.

The guidelines were developed through a review of existing documentation and consolidation within the STARS project. The guideline numbers are derived from corresponding guidelines in *Consolidated Reusability Guidelines* [IBM380]. The numbers were originally derived roughly from the section numbering in the *Reference Manual for the Ada Programming Language* [ADA83].

Coding Style Guidance

Col. Whitaker established the STARS program philosophy with the following note:

> STARS does not wish to impose an excessive or restrictive style on the programmer. A sensible attention to readability and portability should be sufficient guide.
>
> STARS style recommendations have to be consistent with the widest variety of operations, including the thousands of individual shops which may have local ideas, restrictions, and formats enforced by local methods. STARS, therefore, is not restrictive without compelling reason, especially in those areas where it possible to machine restructure the code to any desired style.
>
> STARS sets no specific formatting requirements, as a matter of principle. The philosophy is that one might expect to receive code from various organizations with different ways of doing things. The government will pretty-print to Ada LRM style. The only style limitation is that one should not attempt to encode information (e.g., into the case of identifiers, since Ada is case insensitive), or use other non-Ada conventions. The government should be able to restructure and extract code information that is processable by an Ada compiler.
>
> STARS is trying to develop a software technology to be used by the DoD, not just to control a small group of in-house programmers. The government should not over-specify those things it can easily adapt. Style guidelines that impose more rigid formatting rules are officious pedantry, but very common. Conventions like "_TYPE" may be used by some groups; STARS would not interfere, nor would it attempt to impose them on anyone else.
>
> Arbitrary restrictions to the full capability of Ada (such as unnecessary injunction against "use") are inappropriate. Each local shop may, for its own reasons, add additional restrictions, although STARS would recommend against anything that would limit the expressiveness of Ada. Examples of oppressive limitations include: no "function" in Ada PDL so it can be mapped to COBOL; no "if" nested under an "if", because a tool was derived for a language without "elsif"; forbidding the use of "use", thereby denying much of Ada overloading; forbidding the "while" construct in favor of loops with exit.

STARS is experimenting with using SGML encoding for program prologue information so it can be computer processed. This documentation technique is considered separable from "Ada style", and would be the subject of other guidelines.

Coding Guidelines Summary

The following is a summary of the reusability coding guidelines. It can be used as a checklist during code development.

Design

- 1-1-3 Make cohesion high within each component.
- 1-2-1 Make coupling low.
- 1-2-2 Document each interface thoroughly.
- 1-3-1 Isolate compiler, operating system and machine dependencies.
- 1-3-2 Make all dependent components reusable.

Comments

- 4-1-1 Make each comment adequate, concise and precise.
- 4-2-1 Document each subprogram with a Subprogram Specification Comment Block.

Declarations and Types

General

- 5-1-1 Avoid anonymous types.
- 5-1-2 Try to use limited private types.
- 5-1-3 Use range constraints on numeric types.
- 5-1-5 Avoid predefined and implementation defined types.
- 5-1-6 Explicitly specify the precision required.
- 5-1-7 Use attributes instead of explicit constraints.

Arrays

- 5-2-1 Explicitly declare a type to use in defining discrete ranges.
- 5-2-2 Do not hard code array index designations.

Names

- 6-1-1 Use descriptive identifier names.
- 6-1-2 Keep identifier names less than 80 characters long.
- 6-1-5 Do not overload names from package STANDARD.

Statements

- 7-1-1 Use explicitly declared types for integer ranges in the loop statement.
- 7-2-1 Use elsif for nested if statements.
- 7-3-1 Avoid using the when others clause as a shorthand notation.

Subunits

- 8-1-3 Use named constants for parameter defaults.
- 8-1-7 Use named parameters if there is more than one parameter.
- 8-1-8 Make components complete.
- 8-1-9 Write each module so it has high cohesion.
- 8-1-10 Use information hiding.
  - Only put in the specification those declarations that must be seen externally.
  - Only with compilation units that are really needed.
  - Use private and limited private types to promote information hiding.
- 8-2-1 Use descriptive named constants as return values.

Exceptions

Design

- 9-1-6 Never use the when others construct with the null statement.
- 9-1-7 Avoid pragma suppress.
- 9-1-8 Handle exceptions as close as possible to where they are first raised.
- 9-1-9 Imbed potential elaboration exceptions in a frame.

Propagation

- 9-2-1 Do not propagate an exception beyond where its name is visible
- 9-2-2 Do not propagate predefined exceptions without renaming them.

Usage

- 9-3-2 Do not execute normal control statements from an exception handler.

Documentation

- 9-4-1 Document all exceptions which will be propagated.
- 9-4-2 Clearly list in subprogram specification comment blocks all the conditions that raise exceptions for each operation.

Parameters

- 9-5-1 Be sure that out parameters cannot be undefined.

Implementation dependencies

Design

- 11-1-1 Isolate implementation dependencies.
- 11-1-3 Avoid optional language features.

Pragmas

- 11-2-1 Avoid using pragmas.
- 11-2-2 If pragmas are used, isolate and thoroughly document them.

I/O

- 12-1-1 Encapsulate I/O uses into a separate I/O package.
- 12-2-4 Do not rely on NEW_PAGE.
- 12-2-5 Document implementation dependent procedures.
- 12-3-2 Close files before a program completes.
- 12-5-1 Do not input or output access types.

Coding Guidelines

General Design Guidelines

The following quote sums up the general design issues:

> "Reusability is first and foremost a design issue. If a system is not designed with reusability in mind, component interrelationships will be such that reusability cannot be attained no matter how rigorously coding or documentation rules are followed [AUSNI85]."

Many of the design guidelines listed here are simply good software engineering principles. However, the reverse is not necessarily true; simply following good software engineering principles will not always lead to reusable software [AUSNI85].

Design Guidelines on Generality and Completeness: Designing for generality means making the component easy to adapt to new situations.

Efficiency should be considered when designing for generality. Often algorithms exploiting special properties of a problem may be more efficient than algorithms meant to solve a more general problem.

Completeness means that components should have all functions and operations for current and future needs. Ideally, each component should contain all of the functionality that can be associated with such a component.

Obviously, it is impossible to achieve this for any component, but completeness is still a useful goal. To enhance reusability, components should be made as complete as is practical. Completeness causes development effort to be spent on features not needed for the current project, but probably needed on future projects. It should be tempered by development cost, benefits provided by the component, and likelihood of use.

To achieve generality and completeness, the following guidelines should be followed.

1-1-3 Make Cohesion High Within Each Component: Cohesion is the degree to which the statements in a component form a coherent whole. The most coherent components do just one thing, whether it be manipulating an object or performing a function.

Although not essential for reuse, cohesion is a desirable attribute, because components with high cohesion are likely to be easier to understand and more tailorable, since related code will tend to concentrate in one place.

Cohesion is not measurable, except by inspection. According to [STEVE74] there are several layers of cohesion, listed here from lowest to highest.

Coincidental cohesion	The module does tasks that are related loosely or not at all.
Logical cohesion	The tasks are related in some logical way.
Temporal cohesion	The tasks are related in some way and must be done in the same time span.
Communicational cohesion	All processing elements of a task refer to the same set of input or output data.

Sequential cohesion Output data from one element of the module is input for the next element.

Functional cohesion All elements of a module are related to performing a single function.

In this scheme, low levels of cohesion should be avoided as much as possible. Middle levels of cohesion are about as good as high levels. In practice, it is not necessary to improve the cohesion of a component once it is in the middle range.

[EMBLE87] suggests another way to measure cohesion, by the absence of four strengths -- separable, multifaceted, non-delegation, and concealed. These are defined as follows for abstract data types (ADTs), but the ideas can be generalized to all reusable components.

separable strength An ADT part has separable strength if the part exports an operator (function or procedure) that does not use a domain of the ADT it exports; or the part has a logically exported domain of the ADT that no operator of the part uses; or the part has two or more logically exported domains whose operators do not share any of the domains of the ADT.

multifaceted An ADT part has multifaceted strength if it does not have separable strength, and it exports two or more domains of the ADT. Because it is not separable some operator must share two or more exported domains.

non-delegation An ADT part has non-delegation strength if it has neither separable nor multifaceted strength, and it has an operator that can be delegated to a more primitive ADT.

concealed An ADT part has concealed strength if it has neither separable, multifaceted, nor non-delegation strength and it has a logically hidden ADT.

The above definitions are from [EMBLE87].

1-1-4 Make Components as Complete As Possible: Completeness means that components should have all the functions and operations for current and future needs. Ideally, each component should contain all the functionality that can be associated with such as component. This is, of course, impossible in practice, but minimal guidelines can be established. The following guidelines [SOMM89] concern specifically *object-oriented*components. Each such component should include, either explicitly or implicitly, the following operations.

1. Operations to *create* and *initialize* objects of the abstract type. These operations should be provided explicitly for limited private types, but can be provided implicitly for all other types.
2. Operations to *access* and to *change the value* of each attribute of the implemented object or type.
3. Operations to *assign* objects of the implemented type and to *test for equality*. Again, these should be provided explicitly for limited private types, but can be provided implicitly for all other types.
4. *Test functions* for every exception the component can raise.

Furthermore, if the abstract type is a *composite* type, then the following operations should also be provided.

1. Operations to *add* and *delete* objects from the collection.
2. An *iterator*, which allows each element to be visited.
3. Functions to obtain information about the attributes of the collection as a whole (such as its size).

Design Guidelines on Interfaces: Well-defined interfaces are important for reusability. Below are some guidelines on interface design.

1-2-1 Make Coupling Low: Coupling measures how much modules depend on one another. It depends on the interfaces between modules, the data that pass between them, and the control relationships. Coupling should be as low or loose as possible. This helps make dependencies both clear and isolated, thus making components easier to reuse.

According to [STEVE74] there are several levels of coupling, listed here from lowest to highest.

No coupling	The modules are independent and do not communicate.
Data coupling	Communication is limited to passing simple arguments.
Stamp coupling	A variation of data coupling, where part of a data structure is passed, rather than simple arguments.
Control coupling	Data of a control nature are passed. An example is the passing of a control flag.
External coupling	Modules are tied to specific external environments. For some modules this may be unavoidable, but environment dependence should be isolated as much as possible.
Common coupling	Modules share data in a global data area.
Content coupling	One modules uses the data within the boundary of another module.

In this scheme, coupling should be as low as possible, both for components and for modules making up components. For some modules it may not be possible to achieve the lower levels of coupling (no coupling, data coupling). An effort should be made, however, to build modules with coupling as low as possible in the above scale.

Another way to measure coupling comes from [EMBLE87]. In this scheme two compilation units are *visibly coupled* if one directly accesses the data structures of the other. They are *surreptitiously coupled* if one uses undocumented information about the other's data structures. Finally, they are *loosely coupled* if they are neither visibly nor surreptitiously coupled. In the scheme, the goal is to make components loosely coupled.

1-2-2 Document Each Interface Thoroughly: Well-documented interfaces are important for building reusable components. This will allow programmers to easily understand new code and thus lower the cost of reuse. To thoroughly document interfaces, do the following.

- For generics, explain each formal parameter.
- For subprogram, function and task interfaces, explain each parameter.
- If the interface is unusually complex, describe it thoroughly in a document.

Other Design Guidelines

1-3-2 Make All Dependent Components Reusable: A component is not fully reusable unless all the components it *with*s are reusable. If a component depends on components that are not reusable, then there is a potential for portability and tailorability problems. Thus, when submitting a reusable component to the filtered repository, make sure that all the components it depends on are reusable as well. That is, make sure that each component that is depended on complies with the guidelines in this document.

Comments

General Guidelines

4-1-1 Make Each Comment Adequate, Concise and Precise: This will obviously make the component more readable and thus easier to tailor.

Specific Kinds of Comments: These guidelines recommend the following kinds of comments:

- Exception Documentation Blocks and
- Subprogram Specification Comment Blocks.

Exception documentation blocks are described in "9-4-1 Document All Exceptions Which Will Be Propagated from an Operation in an Exception Documentation Block" on page 14. Subprogram Specification Comment Blocks are described below.

4-2-1 Document Each Subprogram with a Subprogram Specification Comment Block: A subprogram specification should clearly state the intended function of the subprogram. To do this, use a Subprogram Specification Comment Block before each subprogram specification. Give the name of the subprogram and a description of its function. List specific design details, such as conditions that raise exceptions. Be sure to include the conditions that will cause predefined exceptions to be raised and multiple conditions that can cause the same exception.

The comment block could appear with a line of asterisks above or below it. Blank lines might also be placed above the comment block to act as separators. The comment block or part of it could also appear in the body as subprogram commentary. The following example shows a Subprogram Specification Comment Block.

```
--< STRING_TO_DYN_STRING
--<
--< Function:
--<    Return a dynamic string given an Ada string
--< Detail:
--<    if length (ADA_STRING) > MAX_DYNAMIC_STRING_LENGTH then
--<      raise STRING_TOO_LONG
--<    if ADA_STRING = NULL then
--<      return NULL dynamic string
--<    if ADA_STRING /= NULL then
--<      return a dynamic string representation of ADA_STRING
```

Declarations and Types

All the guidelines in this section are designed to improve portability or tailorability.

General Guidelines

5-1-1 Avoid Anonymous Types: An *anonymous type* is a type without a simple name. Consider the following example.

```
Schedule: array (1..5) of Day;
```

Here the type *array (1..5) of Day* has no simple name and is thus an anonymous types. There are only a few cases in Ada where one can create anonymous types; array declarations are one; task declarations are another. One should avoid anonymous types for several reasons:

- There is no self-explanatory type name.
- Using such anonymous types makes qualified expressions impossible. There is no type or subtype to which the programmer can refer in order to qualify the expression.
- Anonymous type impede tailoring because the programmer cannot add assignment statements like the one below without creating a common type beforehand.

```
A,B : array (POSITIVE range MIN .. MAX) of COMPONENT;
begin
    -- Some code
    A := B;        -- Produces a compile-time error

end;
```

5-1-2 Try to Use Limited Private Types: Limited private types help hide design details from the user. Use limited private types when you want neither equality nor assignment exported. If you want these operators to be exported, then use private types instead.

There is one thing that should be kept in mind when using private or limited private types. When limited private or private types are exported, the privacy requirement "propagates." That is, any type that uses a limited private type in its declaration must itself be limited private. Also, any type that uses a private type in its declaration must itself be either private or limited private.

For example, consider the following situation.

- Package A uses a limited private type for the DYN_STRING.

```
type DYN_STRING is limited private;
```

- Package B uses package A and has a data structure that has DYN_STRING as a subcomponent as follows.

```
type NUMBER_RECORD is
    record
      FIELD1 : A.DYN_STRING;
      FIELD2 : STACK;
      FIELD3 : INTEGER;
    end record;
```

- Package C uses package B and refers to the record of package B.

```
type TOTAL is
    record
      ENTRY   : NUMBER_RECORD;
      TALLY_LIST : INTEGER;
    end record;
```

Since package A declares DYN_STRING as a limited private type, then package B must define the NUMBER_RECORD as a limited private type. Package C must use TOTAL as a limited private type because it refers to the NUMBER_RECORD of package B.

This propagation of limited/non-limited private type requirements could cause major rework for packages being modified to use components that use private types. Thus, we believe that a reuse repository should store information on each Ada component on whether the component is based on limited private, private, or non-private types. This would help users select suitable components.

5-1-3 Use Range Constraints on Numeric Types: This causes the compiler to issue a message if the range cannot be supported. The range constraints should be meaningful to the application. [BARNE84]

5-1-5 Avoid Predefined and Implementation Defined Types: Avoid declaring objects of predefined types such as INTEGER. Predefined types are not likely to be portable because their form can vary from Ada implementation to Ada implementation.INTEGER, avoid

5-1-6 Explicitly Specify the Precision Required: Each floating point or fixed point type should explicitly specify the precision, using the **delta** or **digits** accuracy definition. This will make clear any assumptions made about accuracy of calculations.

5-1-7 Use Attributes Instead of Explicit Constraints: Consider the following example from [NISSE84].

```
A: array (DISCRETE_TYPE) of F;
  ...
for I in DISCRETE_TYPE loop
  exit when A(I) < SUM * F'EPSILON;
  SUM := SUM + A(I);
end loop;
```

This example assumes that the series A(1) + A(2) + A(3) + ... converges when all terms are positive. Because the loop depends on F's model numbers and not on explicit constraints, all Ada implementations should have the same accuracy.

Guidelines for Arrays

5-2-1 Explicitly Declare a Type to Use in Defining Discrete Ranges: Use explicitly declared types for discrete ranges. That is, use

```
type DISCRETE_RANGE is range 1..TABLE_SIZE;
type TABLE is array (DISCRETE_RANGE) of ELEMENT_TYPE;
```

instead of

```
type TABLE is array (1..TABLE_SIZE) of ELEMENT_TYPE;
```

[PAPPA85, p. 28]

This provides several benefits. There will be fewer logic errors when components are tailored, because the compiler will have already caught them when it checked for type inconsistencies. Also, the code will be more portable, since the compiler can select the best internal representation for the numeric type requested by the range declaration.

Unfortunately, using explicitly declared types for integer discrete ranges does not always lead to easy to read code. Type conversions may be needed to convert among explicitly declared types. The combination of long type names and required type conversions results in long multi-line Ada statements that are hard to read. Nevertheless, we believe the advantages of using explicitly declared types for integer discrete ranges outweigh the disadvantages.

5-2-2 Do not hard code array index designations: Do not hard code array index designations, as below.

```
type TABLE is array (1..50) of ELEMENT_TYPE;
```

Use types or subtypes instead, because the additional declaration will make the code more self-documenting and thus more tailorable. The upper or lower bound may be an index that will change at some time. The subtype or type declaration will allow the change to be made once instead of many times throughout the program.

Names

6-1-1 Use Descriptive Identifier Names: Use descriptive identifier names to promote readability and self-documentation. Descriptive identifier names make the code clearer. Names should be as long as necessary to provide the needed information and to promote readability. They should be considered part of the documentation of the component.

6-1-2 Keep Identifier Names Less Than 80 Characters Long: Keep identifiers less than 80 characters long, because some Ada implementations use 80 characters as the maximum identifier length. Furthermore, some display devices are limited to 80 characters, since they lack the ability to format larger strings.

6-1-5 Do Not Overload Names from Package STANDARD: Ada names predefined in **package STANDARD** should not be redefined or 'overloaded'. [RYMER86, p. 5] This keeps the reader from confusing the overloaded names with the names predefined in package STANDARD. There is an exception to this rule -- it is permissible to overload the names of operators.

Naming Conventions: Besides the above guidelines, there is no specific naming convention for identifiers in these guidelines. It is assumed that a component retrieval system will exist which will provide tools to analyze component information. The tools will have powerful analytical capabilities, so that a naming convention will not improve the analysis. It will only place

unnecessary constraints on the programmer. However, if the component retrieval tools are not as powerful as anticipated, a naming convention may prove useful, and one should be considered.

Statements

Loop Statement

7-1-1 Use Explicitly Declared Types for Integer Ranges in the Loop Statement: This will improve portability. If no type name is specified, INTEGER is used as the default, which can result in a discrete range being invalid under some Ada implementations. By using type designations, the logic can be more independent of the data.

The following example shows a loop range that should not be used.

```
for I in 1..MAX_NUM_APPLES ...
   ...
end loop;
```

Instead, do the following.

```
type APPLE_COUNT_TYPE is range 1..MAX_NUM_APPLES;
for I in APPLE_COUNT_TYPE loop ...
   ...
end loop;
```

If Statement

7-2-1 Use Elsif for Nested If Statements: This reduces the nesting levels of the **if** statements, giving the code a clean, uncluttered appearance. It also emphasizes the equal status of each **if** statement.

The following is an example from [BARNE84, p. 50] of nested **if** statements.

```
if ORDER = LEFT then
   TURN_LEFT;
else
   if ORDER = RIGHT then
      TURN_RIGHT;
   else
     if ORDER = BACK then
        TURN_BACK;
     end if;
   end if;
end if;
```

The example below shows the above example with **elsifs**.

```
if ORDER = LEFT then
   TURN_LEFT;
elsif ORDER = RIGHT then
   TURN_RIGHT;
elsif ORDER = BACK then
   TURN_BACK;
end if;
```

Case Statement

7-3-1 Avoid Using the When Others Clause as a Shorthand Notation: The **when others** clause of the case statement should not be used as a shorthand to handle all cases that have not been listed. Instead, explicitly handle each case and omit the **when others** clause. If the component is later modified to add more values to the data type, this will call attention to the fact that the new values are not handled in the **case** statement. If the **when others** clause was used, the new data values would be handled by this clause and the operation on the data might be incorrect.

If there is a long list of conditions to be enumerated, use ranges and vertical bars to simplify listing all possible values as in the following example:

```
begin
   case X is
      when AA =>
         -- Some stuff
      when DD =>
         -- Other stuff
      when BB..CC | EE..ZZ =>
         -- The Other other stuff
   end case;
end;
```

Subunits

General Guidelines

8-1-3 Use Named Constants for Parameter Defaults: Use named constants as parameter defaults whenever they would help the reader to better understand the code. For example, this

```
procedure READ (VALUE :     out ELEMENT_TYPE;
                GROUP : in      TAG_GROUP_TYPE := DEFAULT_GROUP);
```

is easier to understand than this.

```
procedure READ (VALUE :     out ELEMENT_TYPE;
                GROUP : in      TAG_GROUP_TYPE := 0);
```

8-1-7 Named Parameters: We do not believe that a set of guidelines should require named parameter association. This should be a user-selectable option with an intelligent formatter. Until such formatters are available, the following are some recommendations on the subject.

1. If there is more than one parameter in the called subprogram, then use named parameter association. This will make the interface clear to the user and make the code self-documenting, particularly when the component user is not supplying all of the possible parameters.

2. If the called subprogram only has one parameter, then use of named parameters is up to the coder. The determining factor should be whether use of the named parameter association will improve readability. Using parameter names in the interface of single parameter function calls particularly hinders readability.

8-1-8 Make Components Complete: Reusable components should be as complete as practical, meaning that the component ideally has all operations to manipulate the given object. For example, a stack package should have such operations as PUSH, POP, CLEAR_STACK and IS_EMPTY. This insures that any stack operation needed in the future will already exist and not need to be coded.

Admittedly, this guideline cannot be fully realized in practice. Yet the goal of completeness is still useful as something to strive for.

The guideline is easier to follow if standard interfaces have been established. For example, there is a standard interface for stack packages, then it will be trivial to inspect a particular stack package to tell whether it provides all the required operations.

8-1-9 Write Each Module So It Has High Cohesion: Cohesion is a measure of the degree to which the statements in a component form a whole. The most coherent components do just one thing, whether it be manipulating an object or performing a function. Cohesion should be maximized whenever possible.

One way to achieve high cohesion is to use an object-oriented design. Such a strategy makes it easy to detect low cohesion. [STDEN86]

8-1-10 Use Information Hiding: There are three guidelines here.

- Only place in the specification section those declarations that must be seen externally.

- Only **with** those compilation units that are really needed. Only if the specification needs such visibility should the context clause appear in the specification; otherwise it should appear in the body. A tool could be written to catch unneeded **withs**.
- Use private and limited private types to promote information hiding.

The rationale comes from the good software engineering practice of minimizing the amount of information visible to the outside world.

Guidelines on Subprograms

8-2-1 Use Descriptive Named Constants as Return Values: Named constants should be returned whenever they would help the reader to understand the code. For example, it is more informative to return the named constant NOT_FOUND than to return the value -1.

Exception Handling

As stated earlier in this document, good exception handling is important to software reuse for several reasons.

- Components with good error / exception handling have safety built in.
- Errors are isolated and well-documented.
- The way interfaces work is made clear. There are fewer hidden assumptions.
- The users have the freedom to decide whether to propagate exceptions further, to retry the operation that raised the exception, to abandon the operation, or to continue regardless.
- Good exception handling makes components more tailorable and thus more reusable.

Exception Handling Design

9-1-6 Avoid Using the When Others Construct with the Null Statement: Use of the **null** statement suggests that the exception is not used for an abnormal condition.

```
begin
  loop
    ...
    raise MISCELLANEOUS_ERROR;
    ...
  end;
exception
 when others =>
  null;
 end;
 -- rest of normal program code
```

In the above example a **raise** statement is used to exit the loop and to continue executing normal control flow. This implies that there never was an abnormal condition.

9-1-7 Avoid pragma SUPPRESS: The Ada Language Reference Manual [ADA83] does not require that **pragma SUPPRESS** be implemented. **Pragma SUPPRESS** does not guarantee that exceptions will not be propagated to a unit for which exception suppression is in effect. The execution of a program is erroneous if an exception occurs while **pragma SUPPRESS** is in effect.

9-1-8 Handle Exceptions as Close as Possible to Where They Are First Raised: This gives the exception handler access to local data, which can be used to respond to the exception. It also avoids losing visibility to the exception name.

When an exception is propagated to a scope outside its visibility, its name is lost. The exception can only be handled by a **when others** handler. Such exceptions might be unwittingly handled by a handler that was never intended to handle the exception.

Below is an example of an exception handler making use of local data.

```
package body SEQUENTIAL_ACCESS_METHOD is

   CURRENT_RECORD : RECORD_IDENTIFIER := START_OF_FILE;

   procedure GET (FILE : in FILE_TYPE;
                  REC : out RECORD_TYPE) is
      VALUE : RECORD_TYPE;
   begin
      ...
      VSAM.VGET(CATALOG => FILE,
                INDEX   => CURRENT_RECORD + 1,
                DATA    => VALUE);
      ...
      exception
        when DEVICE_ERROR =>
          ERROR_IO.LOG("Error Occurred Reading Record"
amp
                    RECORD_IDENTIFIER'IMAGE(CURRENT_RECORD + 1));
   end GET;
end SEQUENTIAL_ACCESS_METHOD;
```

9-1-9 Imbed Potential Elaboration Exceptions in a Frame: Unless special provisions are made, elaboration exceptions are not handled in the unit being elaborated. Consider the following example.

```
package ELABORATION_EXCEPTION_PKG is
  S : STRING (1..2) := "Causes Constraint_Error";
end ELABORATION_EXCEPTION_PKG;
```

The Constraint_Error exception that is generated is not handled inside the package; it is propagated out.

The solution is to imbed potential elaboration exceptions in a frame. To do this, do the following.

- Move declarations to declare blocks inside executable regions,
- Do initializations inside executable regions, and
- Encapsulate initializations within a subprogram to take advantage of Ada's strong typing, as in the following example.

```
with INITIALIZATION_PKG;
package ELABORATION_EXCEPTION_PKG is
  S : INITIALIZATION_PKG.NAME_TYPE := INITIALIZATION_PKG.SET_NAME;
end ELABORATION_EXCEPTION_PKG;

package INITIALIZATION_PKG is
   subtype NAME_TYPE is STRING(1..2);
   function SET_NAME return NAME_TYPE;
     -- guarantees no CONSTRAINT_ERROR
end INITIALIZATION_PKG;
```

Exception Propagation

9-2-1 Do Not Propagate an Exception Beyond Where Its Name Is Visible: Do not propagate an exception beyond where its name is visible. Otherwise, it can only be handled by a **when others** handler.

9-2-2 Do Not Propagate Predefined Exceptions Without Renaming Them: Predefined exceptions have no corresponding **raise** statement in the source code, so it is not always obvious that an exception can be propagated. Predefined exceptions can be raised by many operations, making them difficult to locate. Renaming predefined expressions makes it easier to pinpoint the exact cause of each exception. For example, the predefined exception STORAGE_ERROR might be propagated as MEMORY_FULL.

Use of Exception Handling

9-3-2 Do Not Execute Normal Control Statements from an Exception Handler: Only use exception handling for abnormal control flow, not for normal control.

Below is an example of poor use of exception handling.

```
begin
  loop
    TEXT_IO.GET(DATA_FILE,DATA_VALUE);
    ...
  end loop;
exception
  when TEXT_IO.END_ERROR(DATA_FILE) =>
      -- execute the rest of the program here
end;
```

In contrast, the following shows equivalent code without the use of an exception handler.

```
while not TEXT_IO.END_OF_FILE(DATA_FILE) loop
   TEXT_IO.GET(DATA_FILE,DATA_VALUE);
   ...
end loop;
-- execute the rest of the program here
```

Exception Documentation

9-4-1 Document All Exceptions Which Will Be Propagated from an Operation in an Exception Documentation Block: An Exception Documentation Block shows which operations raise which exceptions under what conditions. In this block, describe all the conditions that cause each exception to be raised, including predefined exceptions. This will help other developers in making their designs complete.

Be sure to clearly associate each exception with every operation where the exception can be raised. If the same operation can raise an exception for different reasons, record each reason separately.

The following is an example of an Exception Documentation Block.

```
STRING_TOO_LONG : exception;
   ------------------------------------------------------------------
   --
   --   Raised By              On Condition
   --   ---------              ------------
   --
   --   INSERT                 the size of the string with the
   --                          insertion exceeds MAX_DYNAMIC_
   --                          STRING_LENGTH
   --
   --   REPLACE                the size of the string with the
   --                          replaced part exceeds MAX_DYNAMIC_
   --                          STRING_LENGTH
   ------------------------------------------------------------------
```

9-4-2 Clearly List in Subprogram Specification Comment Blocks All the Conditions That Raise Exceptions For Each Operation: This includes the conditions that will cause predefined exceptions to be raised and includes multiple conditions that can cause the same exception.

Exception Handling Parameter Usage

9-5-1 Be Sure That Out Parameters Cannot Be Undefined Upon Return from a Subprogram If an Exception Occurs: Never depend on the value of **out** parameters or **return** values when designing a handler response. When an exception occurs while evaluating the right side of an expression, then the current value of the variable stays the same. The values of scalar **out** parameters which are not updated are undefined. Thus, the exception handler should set the values of scalar parameters before returning.

Implementation Dependencies

Design Considerations

11-1-1 Isolate Compiler, Operating System and Machine Dependencies dependencies*:* To make components portable, avoid optional language features and Ada implementation dependencies. Where this cannot be done, isolate such uses, so users can plug in new versions easily. Document all such uses. Both encapsulation and documentation will reduce the effort to port a component to a new implementation.

Write code to ignore details of underlying implementations. Components should be designed without reference to the surrounding environment. Contact between a component and its environment should occur through explicit parameters and explicitly invoked subprograms. [PAPPA85, p. 7]

11-1-3 Avoid Optional Language Features: For example, avoid using UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION. These two procedures are optional and implementation dependent. If you use these procedures, document their use. *Environment_Imposed_Restrictions* and *Compiler_Dependent_Restrictions.*

Pragmas

11-2-1 Avoid Using Pragmas: Pragmas are generally environment dependent. Sometimes, though, their use may be unavoidable.

Pragma INTERFACE may be needed to specify interfaces with subprograms of other languages. Pragma ELABORATE may be needed to insure that a program is correctly elaborated no matter what compiler is used, since elaboration order varies from compiler to compiler. However, take care that pragma ELABORATE is essential and not needed because the component is overly complex.

11-2-2 If Pragmas Are Used, Isolate and Thoroughly Document Them: If they must be used, they should be isolated as much as possible.

Those components which use pragmas should be documented, pointing them out and describing their effects. *Pragmas.*

For example, use of **pragma INLINE** in a reusable component should be documented. Its use can force the user's code to depend on the body of the reusable component. Since this effect is usually unexpected, take care to insure that the reuser is aware of the compilation issues caused by it.

Input and Output

General Guidelines

12-1-1 Encapsulate I/O Uses Into a Separate I/O Package: All input/output utilities should be isolated into I/O packages. This will make it easy for users to adapt the component to different machines and operating systems.

Guidelines on Specific I/O Procedures and Functions

12-2-4 Do Not Rely on NEW_PAGE: The Ada language standard does not specify the value of a page terminator. Thus, the control characters may be non-portable across printers. One solution is to always direct output to a file, which can then be filtered and altered to suit the device the output is ultimately destined for.

12-2-5 Document Implementation Dependent Procedures: Use of the following procedures could result in portability problems. Dependencies on such procedures should be documented. *Portability_Restrictions.*

- COL -- Depends on the implementation-defined subtype POSITIVE_COUNT.

- DIRECT_IO.READ -- Reads from an index whose range POSITIVE_COUNT is implementation defined. The DIRECT_IO.WRITE procedure could also cause similar problems.
- ENUMERATION_IO.GET -- Returns an **out** parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input. It also has a WIDTH parameter of the implementation-defined type FIELD.
- FIXED_IO.GET -- Returns an **out** parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input. It also has a WIDTH parameter of the implementation-defined type FIELD.
- FIXED_IO.PUT -- Has a WIDTH parameter of the implementation-defined type FIELD.
- FLOAT_IO.GET -- Returns an **out** parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input. It also has a WIDTH parameter of the implementation-defined type FIELD.
- FLOAT_IO.PUT -- Has a WIDTH parameter of the implementation-defined type FIELD.
- GET_LINE -- Returns an **out** parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input.
- INDEX -- Uses an index whose range POSITIVE_COUNT is implementation-defined.
- INTEGER_IO.GET -- Returns an **out** parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input. It also has a WIDTH parameter of the implementation-defined type FIELD.
- INTEGER_IO.PUT -- Has a WIDTH parameter of the implementation-defined type FIELD.
- LINE -- Depends on the implementation-defined subtype POSITIVE_COUNT.
- LINE_LENGTH -- Depends on the implementation-defined type COUNT, whose upper bound varies with each implementation.
- PAGE -- Depends on the implementation-defined subtype POSITIVE_COUNT.
- PAGE_LENGTH -- Depends on the implementation-defined type COUNT, whose upper bound varies with each implementation.
- SET_COL -- Depends on the implementation-defined subtype POSITIVE_COUNT.
- SET_LINE -- Depends on the implementation-defined subtype POSITIVE_COUNT.
- SET_INDEX -- Uses an index whose range POSITIVE_COUNT is implementation-defined.
- SET_LINE_LENGTH -- Depends on the implementation-defined type COUNT, whose upper bound varies with each implementation.
- SET_PAGE_LENGTH -- Depends on the implementation-defined type COUNT, whose upper bound varies with each implementation.
- SIZE -- Uses an index whose range POSITIVE_COUNT is implementation-defined.

[MATTH87b, p. 2]

Furthermore, using procedures SKIP_LINE and NEW_LINE to skip more than one line at a time may lead to portability problems, since they depend on the implementation-defined subtype POSITIVE_COUNT. Skipping one line will not cause any problems; however, skipping multiple lines may not be portable depending on the constraint set by the Ada implementation. [MATTH87b, p. 2]

File Handling

12-3-2 Close Files Before a Program Completes: Different Ada implementations handle unclosed files in different ways. The state of unclosed files after program termination is undefined. To increase the reusability of a component, close all files before a subprogram terminates normally or abnormally. Be sure to verify that a file is open before closing it so that the exception STATUS_ERROR is not raised.

I/O of Access Types

12-5-1 Do Not Input or Output Access Types: The effect of I/O of access types is undefined. If used, it may lead to components that are not portable. To output an object pointed to, output the object. To output the address of an object pointed to, output the address of the object using SYSTEM.ADDRESS. [MATTH87b, p. 1] Document the use of SYSTEM.ADDRESS.

References

There are numerous coding guidelines available, particularly for Ada. The following is a list of references for the STARS reusability coding guidelines.

[ADA83] *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, February, 17 1983.

[AHO74] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Readding, Mass.: Addison-Wesley, 1974.

[AUSNI85] Ausnit, Christine, Christine Braun, Sterling Eanes, John Goodenough, Richard Simpson, *Ada Reusability Guidelines*, SofTech, Inc., April 1985.

[BARNE84] Barnes, J.G.P., *Programming in Ada*, 2nd edition. Addison-Wesley Publishers Limited, 1984.

[BENTL85] Bentley, Jon, "Programming Pearls," *Communications of the ACM*, vol. 28, no. 7 July 1985.

[BOOCH87] Booch, Grady, *Software Components With Ada.* The Benjamin/Cummings Publishing Company, Inc., 1987.

[EMBLE87] Embley, David W. and Woodfield, Scott N., "Cohesion and Coupling for Abstract Data Types." *Proceedings, Sixth Phoenix Conference on Computers and Communications*, Phoenix, Arizona, February 1987.

[EVBSE87] EVB Software Engineering, Inc., *Creating Reusable Ada Software*, 1987.

[IBM0340] IBM Systems Integration Division, *Informal Technical Report on Findings During the Rebuild of Common Capabilities*, CDRL Sequence No. 0340, February 19, 1989.

[IBM0360] IBM Systems Integration Division, *Reusability Guidelines*, CDRL Sequence No. 0360, December 17, 1988.

[IBM0370] IBM Systems Integration Division, *Reusable Component Data Analysis*, CDRL Sequence No. 0370, February 10, 1989.

[IBM0380] IBM Systems Integration Division, *Consolidated Reusability Guidelines*, CDRL Sequence No. 0380, March 21, 1989.

[IBM0460] IBM Systems Integration Division, *Repository Guidelines and Standards*, CDRL Sequence No. 0460, March 17, 1989.

[IBM0520] IBM Systems Integration Division, *Long Term Configuration Management Plan for the STARS Repository*, CDRL Sequence No. 0520, March 17, 1989.

[IBM0710] IBM Systems Integration Division, *DTD Definition: Internal Documentation*, CDRL Sequence No. 0710, January 16, 1989.

[MATSU84] Matsumoto, Y., "Some Experiences in Promoting Reusable Software Presentation in Higher Abstract Levels," *IEEE Transactions on Software Engineering*, vol. SE-10 (5), September 1984.

[MATTH87a] Matthews, E. R., *IBM Federal Systems Division Guide for Reusable Ada Components (Draft)*, September 17, 1987.

[MATTH87b] Matthews, E. R., "Observations on the Portability of Ada I/O", *ACM SIGAda Letters*, vol. VII, no. 5, September/October 1987.

[MCILR68] McIlroy, M. D., "Mass Produced Software Components," *Report on a conference by the NATO Science Committee*, Garmisch, Germany, October 7-11, 1968.

[MENDA88] Mendal, Geoffrey O., "Three Reasons to Avoid the Use Clause," *ACM SIGAda Letters*, vol. VIII, no. 1, January/February 1988.

[NISSE84] Nissen, John and Peter Wallis, *Portability and Style in Ada*, Cambridge University Press, 1984.

[PAPPA85] Pappas, Frank, *Ada Portability Guidelines*, SofTech, Inc., March 1985.

[RACIN88] Racine, Roger, "Why the Use Clause is Beneficial," *ACM SIGAda Letters*, vol. VIII, no. 3, May/June 1988).

[ROSEN87] Rosen, J. P., "In defense of the 'use' clause," *ACM SIGAda Letters*, vol. VII, no. 7, November/December 1987.

[RYMER86] Rymer, John and McKeever, Tom., *The FSD Ada Style Guide*, 1986.

[SOMM89] Sommerville, I., *Software Engineering*, 3rd. edition, Addison-Wesley, 1989.

[STDEN86] St. Dennis, R., P. Stachour, E. Frankowski, and E. Onuegbe, "Measurable Characteristics of Reusable Ada Software," *ACM SIGAda Ada Letters*, vol. VI, no. 2, March/April 1986.

[STEVE74] Stevens, W. P., G. J. Myers, and L. L. Constantine, "Structured design." *IBM Systems Journal*, 1974, no. 2.

[UNISYS0340] Unisys Corporation, *Draft Technical Report on Reusability Guidelines*, CDRL 0340, February 14, 1989.

Glossary

The following terms and definitions describe component attributes and design issues, as used in these guidelines.

anonymous type: A type without a simple name.

cohesion: A measure of the degree to which the code in a module forms a coherent whole.

contact: The contact is the person in the producing company who is the 'point of contact' for that particular part or product. Point of contact refers to the person who is familiar with the product, and can either answer questions about it, or can refer people to someone who can answer them.

coupling: A measure of how much components or modules depend on each other. Coupling depends on the interfaces between modules, the data that pass between them, and the control relationships.

dynamic stack: The stack of calls made at runtime.

exception documentation block: A comment that documents an exception.

frame: An Ada language construct that surrounds an exception handler. A frame can be a block statement or the body of a subprogram, a package, a task, or a generic.

functional completeness: The idea that components should have all functions and operations required for current and future needs.

independence: The ability of a component to be used with different compilers, operating systems, machines and applications than those for which it was originally developed. Independence is closely related to *portability*.

maintainability: The ease of modifying a component, whether it be to meet particular needs or to fix bugs.

order (of an algorithm): A measure of the computational efficiency of an algorithm, expressed in terms of the frequency of some key operation. For more information, see [AHO74].

overloading: The property whereby Ada literals, operators, identifiers, and aggregates can have unambiguous alternative meanings.

platform: Platform refers to the architecture for the system for which the product is intended (hardware, operating system, and Ada compiler). Some products may be intended for several different platforms. Platforms listed should also indicate whether they are host platforms, target platforms, or both.

portability: The ability of an application or component to be used again in a different target environment than the one it was originally built for. The phrase *target environment* may be defined broadly to include operating systems, machines, and applications. To be ported effectively, components may need to be tailored to the requirements of the new target environment. See also *reusability* and *independence.*

reliability: The extent to which a component performs as specified. A reusable component performs consistently with repeated use and across environments (that is, operating systems and hardware).

reusability: The ability to reuse a software component or to use it repeatedly in applications other than the one for which it was originally built. In order to be effectively reused, the component may have to be tailored to the requirements of the new application. See also *portability*.

subprogram specification comment block: A comment block that accompanies a subprogram specification, giving its name and a description of its function.

tailorability: The ease of modifying a component to meet particular needs. It should be distinguished from *maintainability*, which includes tailorability, but also includes the idea of corrective maintenance (fixing bugs).

Index

A

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

STARS

Reusability Guidelines

April 30, 1990

Contract No. F19628-88-D-0032

Task IR40: Repository Integration

Delivered as part of:
CDRL Sequence No. 1550

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscomb AFB, MA 01731-5000

Prepared by:

IBM Systems Integration Division
800 North Frederick
Gaithersburg, MD 20879

Abstract

This document describes the STARS project Ada coding guidelines, as they relate to reusability. This document is part of a suite of documents that define the use of the IBM STARS Repository. The other documents are the *IBM STARS Repository GuideBook* and the *IBM STARS Repository User's Guide*

Preface

The guidelines in this document apply to all IBM STARS team tasks. In particular, they apply to Ada code and related documentation as it is admitted to and managed within the IBM STARS Repository.

These guidelines were originally published in the *Consolidated Reusability Guidelines* IBM0380. Essentially, the guidelines are the same as the guidelines which were collectively established by the STARS prime contractors in the STARS Q-Increment. Some of the original guidelines were modified for clarity and depth of definition. A few were eliminated due to experiences and comments since they were published. The guidelines have also been adjusted to match the metrics produced by the Repository metric collection tool.

This document was developed by the IBM Systems Integration Division, located at 800 North Frederick, Gaithersburg, MD 20879. Questions or comments should be directed to the author, Robert W. Ekman at (301) 240-6431, or to the IBM STARS Program Office.

Contents

1. Introduction

This document presents a set of Ada coding guidelines for component development that emphasize reusability. By following these guidelines, your code will be easier to reuse across multiple projects and platforms. Compliance to these guidelines will increase the level of acceptance within the Repository component evaluation process.

Where appropriate, the following information is provided:

- A brief statement of the guideline,
- A detailed explanation of its meaning,
- A rationale on why the guideline is needed,
- Suggestions on how to apply the guideline, and
- Examples to illustrate the use of the guideline.

The guidelines were developed through a review of existing documentation and consolidation within the STARS project. The guidelines are numbered consecutively by topics. The topics correspond roughly and are defined in the same order as the the section numbering in the *Reference Manual for the Ada Programming Language* [ADA83].

Coding Style Guidance

Col. Whitaker established the STARS program philosophy with the following note:

> STARS does not wish to impose an excessive or restrictive style on the programmer. A sensible attention to readability and portability should be sufficient guide.
>
> STARS style recommendations have to be consistent with the widest variety of operations, including the thousands of individual shops which may have local ideas, restrictions, and formats enforced by local methods. STARS, therefore, is not restrictive without compelling reason, especially in those areas where it possible to machine restructure the code to any desired style.
>
> STARS sets no specific formatting requirements, as a matter of principle. The philosophy is that one might expect to receive code from various organizations with different ways of doing things. The government will pretty-print to Ada LRM style. The only style limitation is that one should not attempt to encode information (e.g., into the case of identifiers, since Ada is case insensitive), or use other non-Ada conventions. The government should be able to restructure and extract code information that is processable by an Ada compiler.
>
> STARS is trying to develop a software technology to be used by the DoD, not just to control a small group of in-house programmers. The government should not over-specify those things it can easily adapt. Style guidelines that impose more rigid formatting rules are officious pedantry, but very common. Conventions like "_TYPE" may be used by some groups; STARS would not interfere, nor would it attempt to impose them on anyone else.
>
> Arbitrary restrictions to the full capability of Ada (such as unnecessary injunction against "use") are inappropriate. Each local shop may, for its own reasons, add additional restrictions, although STARS would recommend against anything that would limit the expressiveness of Ada. Examples of oppressive limitations include: no "function" in Ada PDL so it can be mapped to COBOL; no "if" nested under an "if," because a tool was derived for a language without "elsif"; forbidding the use of "use," thereby denying much of Ada overloading; forbidding the "while" construct in favor of loops with exit.

STARS is experimenting with using SGML encoding for program prologue information so it can be computer processed. This documentation technique is considered separable from "Ada style", and would be the subject of other guidelines.

Coding Guidelines Summary

The following is a summary of the reusability coding guidelines. It can be used as a checklist during code development.

Design

- DES-1 Make cohesion high within each component.
- DES-2 Make components as complete as possible.
- DES-3 Make coupling low.
- DES-4 Document each interface thoroughly.
- DES-5 Make all dependent components reusable.
- IMP-1 Isolate compiler, operating system and machine dependencies.

Comments

- COM-1 Make each comment adequate, concise and precise.
- COM-2 Document each subprogram with a Subprogram Specification Comment Block.

Declarations and Types

General

- DEC-1 Avoid anonymous types.
- DEC-2 Try to use limited private types.
- DEC-3 Use range constraints on numeric types.
- DEC-4 Avoid predefined and implementation defined types.
- DEC-5 Explicitly specify the precision required.
- DEC-6 Use attributes instead of explicit constraints.
- DEC-8 Declare invariants as constants
- DEC-9 Initialize variables during declaration

Arrays

- ARR-1 Explicitly declare a type to use in defining discrete ranges.
- ARR-2 Do not hard code array index designations.

Names

- NAM-1 Use descriptive identifier names.
- NAM-2 Keep identifier names less than 80 characters long.
- NAM-3 Do not overload names from package STANDARD.

Statements

- STA-1 Use explicitly declared types for integer ranges in the loop statement.
- STA-2 Exit Basic LOOPs only via Conditional
- STA-3 Use elsif for nested if statements.
- STA-4 Avoid using the when others clause as a shorthand notation.

Subunits

- SUB-1 Use named constants for parameter defaults.
- SUB-2 Use named parameters if there is more than one parameter.
- SUB-3 Make components complete.
- SUB-4 Write each module so it has high cohesion.
- SUB-5 Use information hiding.

- Only put in the specification those declarations that must be seen externally.
- Only with compilation units that are really needed.
- Use private and limited private types to promote information hiding.

- SUB-6 Use descriptive named constants as return values.

Exceptions

Design

- EXP-1 Avoid the when others construct with the null statement.
- EXP-2 Avoid pragma suppress.
- EXP-3 Handle exceptions close to where they are first raised.
- EXP-4 Imbed potential elaboration exceptions in a frame.

Propagation

- EXP-5 Do not propagate an exception where its name is not visible
- EXP-6 Do not propagate predefined exceptions without renaming them.

Usage

- EXP-7 Do not execute normal control statements from an exception handler.

Documentation

- EXP-8 Document all exceptions which will be propagated.
- EXP-9 List all Conditions That Raise Exceptions in Subprogram Specification Comment Blocks

Parameters

- EXP-10 Ensure that out parameters cannot be undefined.

Implementation dependencies

Design

- IMP-1 Isolate implementation dependencies.
- IMP-2 Avoid optional language features.

Pragmas

- IMP-3 Avoid using pragmas.
- IMP-4 If pragmas are used, isolate and thoroughly document them.

I/O

- IO-1 Encapsulate I/O uses into a separate I/O package.
- IO-2 Do not rely on NEW_PAGE.
- IO-3 Document implementation dependent procedures.
- IO-4 Close files before a program completes.
- IO-5 Do not input or output access types.

2. Coding Guidelines

General Design Guidelines

The following quote sums up the general design issues:

> Reusability is first and foremost a design issue. If a system is not designed with reusability in mind, component interrelationships will be such that reusability cannot be attained no matter how rigorously coding or documentation rules are followed [AUSNI85.]

Many of the design guidelines listed here are simply good software engineering principles. However, the reverse is not necessarily true; simply following good software engineering principles will not always lead to reusable software [AUSNI85].

Design Guidelines on Generality and Completeness: Designing for generality means making the component easy to adapt to new situations.

Efficiency should be considered when designing for generality. Often algorithms exploiting special properties of a problem may be more efficient than algorithms meant to solve a more general problem.

Completeness means that components should have all functions and operations for current and future needs. Ideally, each component should contain all of the functionality that can be associated with such a component.

Obviously, it is impossible to achieve this for any component, but completeness is still a useful goal. To enhance reusability, components should be made as complete as is practical. Completeness causes development effort to be spent on features not needed for the current project, but probably needed on future projects. It should be tempered by development cost, benefits provided by the component, and likelihood of use.

To achieve generality and completeness, the following guidelines should be followed.

DES-1 Make Cohesion High Within Each Component: Cohesion is the degree to which the statements in a component form a coherent whole. The most coherent components do just one thing, whether it be manipulating an object or performing a function.

Although not essential for reuse, cohesion is a desirable attribute, because components with high cohesion are likely to be easier to understand and more tailorable, since related code will tend to concentrate in one place.

Cohesion is not measurable, except by inspection. According to [STEVE74] there are several layers of cohesion, listed here from lowest to highest.

Coincidental cohesion	The module does tasks that are related loosely or not at all.
Logical cohesion	The tasks are related in some logical way.
Temporal cohesion	The tasks are related in some way and must be done in the same time span.
Communicational cohesion	All processing elements of a task refer to the same set of input or output data.
Sequential cohesion	Output data from one element of the module is input for the next element.
Functional cohesion	All elements of a module are related to performing a single function.

In this scheme, low levels of cohesion should be avoided as much as possible. Middle levels of cohesion are about as good as high levels. In practice, it is not necessary to improve the cohesion of a component once it is in the middle range.

[EMBLE87] suggests another way to measure cohesion, by the absence of four strengths -- separable, multifaceted, non-delegation, and concealed. These are defined as follows for abstract data types (ADTs), but the ideas can be generalized to all reusable components.

separable strength An ADT part has separable strength if the part exports an operator (function or procedure) that does not use a domain of the ADT it exports; or the part has a logically exported domain of the ADT that no operator of the part uses; or the part has two or more logically exported domains whose operators do not share any of the domains of the ADT.

multifaceted An ADT part has multifaceted strength if it does not have separable strength, and it exports two or more domains of the ADT. Because it is not separable some operator must share two or more exported domains.

non-delegation An ADT part has non-delegation strength if it has neither separable nor multifaceted strength, and it has an operator that can be delegated to a more primitive ADT.

concealed An ADT part has concealed strength if it has neither separable, multifaceted, nor non-delegation strength and it has a logically hidden ADT.

The above definitions are from [EMBLE87].

DES-2 Make Components as Complete As Possible: Completeness means that components should have all the functions and operations for current and future needs. Ideally, each component should contain all the functionality that can be associated with such as component. This is, of course, impossible in practice, but minimal guidelines can be established. The following guidelines [SOMM89] concern specifically *object-oriented*components. Each such component should include, either explicitly or implicitly, the following operations.

1. Operations to *create* and *initialize* objects of the abstract type. These operations should be provided explicitly for limited private types, but can be provided implicitly for all other types.
2. Operations to *access* and to *change the value* of each attribute of the implemented object or type.
3. Operations to *assign* objects of the implemented type and to *test for equality*. Again, these should be provided explicitly for limited private types, but can be provided implicitly for all other types.
4. *Test functions* for every exception the component can raise.

Furthermore, if the abstract type is a *composite* type, then the following operations should also be provided.

1. Operations to *add* and *delete* objects from the collection.
2. An *iterator*, which allows each element to be visited.
3. Functions to obtain information about the attributes of the collection as a whole (such as its size).

Design Guidelines on Interfaces: Well-defined interfaces are important for reusability. Below are some guidelines on interface design.

DES-3 Make Coupling Low: Coupling measures how much modules depend on one another. It depends on the interfaces between modules, the data that pass between them, and the control relationships. Coupling should be as low or loose as possible. This helps make dependencies both clear and isolated, thus making components easier to reuse.

According to [STEVE74] there are several levels of coupling, listed here from lowest to highest.

No coupling	The modules are independent and do not communicate.
Data coupling	Communication is limited to passing simple arguments.
Stamp coupling	A variation of data coupling, where part of a data structure is passed, rather than simple arguments.
Control coupling	Data of a control nature are passed. An example is the passing of a control flag.
External coupling	Modules are tied to specific external environments. For some modules this may be unavoidable, but environment dependence should be isolated as much as possible.
Common coupling	Modules share data in a global data area.
Content coupling	One modules uses the data within the boundary of another module.

In this scheme, coupling should be as low as possible, both for components and for modules making up components. For some modules it may not be possible to achieve the lower levels of coupling (no coupling, data coupling). An effort should be made, however, to build modules with coupling as low as possible in the above scale.

Another way to measure coupling comes from [EMBLE87]. In this scheme two compilation units are *visibly coupled* if one directly accesses the data structures of the other. They are *surreptitiously coupled* if one uses undocumented information about the other's data structures. Finally, they are *loosely coupled* if they are neither visibly nor surreptitiously coupled. In the scheme, the goal is to make components loosely coupled.

DES-4 Document Each Interface Thoroughly: Well-documented interfaces are important for building reusable components. This will allow programmers to easily understand new code and thus lower the cost of reuse. To thoroughly document interfaces, do the following.

- For generics, explain each formal parameter.
- For subprogram, function and task interfaces, explain each parameter.
- If the interface is unusually complex, describe it thoroughly in a document.

Portability and Design

DES-5 Make All Dependent Components Reusable: A component is not fully reusable unless all the components it *with*s are reusable. If a component depends on components that are not reusable, then there is a potential for portability and tailorability problems. Thus, when submitting a reusable component to the filtered repository, make sure that all the components it depends on are reusable as well. That is, make sure that each component that is depended on complies with the guidelines in this document.

Comments

General Guidelines

COM-1 Make Each Comment Adequate, Concise and Precise: This will obviously make the component more readable and thus easier to tailor.

Specific Kinds of Comments: These guidelines recommend the following kinds of comments:

- Exception Documentation Blocks and
- Subprogram Specification Comment Blocks.

Exception documentation blocks are described in <hdref refid=hexbloc>. Subprogram Specification Comment Blocks are described below.

COM-2 Document Each Subprogram with a Subprogram Specification Comment Block*:* A subprogram specification should clearly state the intended function of the subprogram. To do this, use a Subprogram Specification Comment Block before each subprogram specification. Give the name of the subprogram and a description of its function. List specific design details, such as conditions that raise exceptions. Be sure to include the conditions that will cause predefined exceptions to be raised and multiple conditions that can cause the same exception.

The comment block could appear with a line of asterisks above or below it. Blank lines might also be placed above the comment block to act as separators. The comment block or part of it could also appear in the body as subprogram commentary. The following example shows a Subprogram Specification Comment Block.

```
--< STRING_TO_DYN_STRING
--<
--< Function:
--<    Return a dynamic string given an Ada string
--< Detail:
--<    if length (ADA_STRING) > MAX_DYNAMIC_STRING_LENGTH then
--<      raise STRING_TOO_LONG
--<    if ADA_STRING = NULL then
--<      return NULL dynamic string
--<    if ADA_STRING /= NULL then
--<      return a dynamic string representation of ADA_STRING
```

Declarations and Types

All the guidelines in this section are designed to improve portability or tailorability.

General Guidelines

DEC-1 Avoid Anonymous Types: An *anonymous type* is a type without a simple name. Consider the following example.

```
Schedule: array (1..5) of Day;
```

Here the type *array (1..5) of Day* has no simple name and is thus an anonymous types. There are only a few cases in Ada where one can create anonymous types; array declarations are one; task declarations are another. One should avoid anonymous types for several reasons:

- There is no self-explanatory type name.
- Using such anonymous types makes qualified expressions impossible. There is no type or subtype to which the programmer can refer in order to qualify the expression.
- Anonymous type impede tailoring because the programmer cannot add assignment statements like the one below without creating a common type beforehand.

```
A,B : array (POSITIVE range MIN .. MAX) of COMPONENT;
begin
   -- Some code
   A := B;       -- Produces a compile-time error

end;
```

DEC-2 Try to Use Limited Private Types: Limited private types help hide design details from the user. Use limited private types when you want neither equality nor assignment exported. If you want these operators to be exported, then use private types instead.

There is one thing that should be kept in mind when using private or limited private types. When limited private or private types are exported, the privacy requirement "propagates." That is, any type that uses a limited private type in its declaration must itself be limited private. Also, any type that uses a private type in its declaration must itself be either private or limited private.

For example, consider the following situation.

- Package A uses a limited private type for the DYN_STRING.

```
type DYN_STRING is limited private;
```

- Package B uses package A and has a data structure that has DYN_STRING as a subcomponent as follows.

```
type NUMBER_RECORD is
    record
      FIELD1 : A.DYN_STRING;
      FIELD2 : STACK;
      FIELD3 : INTEGER;
    end record;
```

- Package C uses package B and refers to the record of package B.

```
type TOTAL is
    record
      ENTRY  : NUMBER_RECORD;
      TALLY_LIST : INTEGER;
    end record;
```

Since package A declares DYN_STRING as a limited private type, then package B must define the NUMBER_RECORD as a limited private type. Package C must use TOTAL as a limited private type because it refers to the NUMBER_RECORD of package B.

This propagation of limited/non-limited private type requirements could cause major rework for packages being modified to use components that use private types. Thus, we believe that a reuse repository should store information on each Ada component on whether the component is based on limited private, private, or non-private types. This would help users select suitable components.

DEC-3 Use Range Constraints on Numeric Types: For example, instead of

```
type t1 is digits 5;
```

code the following:

```
type t1 is digits 5 range 0.0 .. 100.0;
```

This causes the compiler to issue a message if the range cannot be supported. The range constraints should be meaningful to the application. [BARNE84]

DEC-4 Avoid Predefined and Implementation Defined Types: Avoid declaring objects of predefined types such as INTEGER, and implementation types such as LONG_INTEGER (from appendix F of each ADA implementation). Predefined and implementation defined types are not likely to be portable because their form can vary from Ada implementation to Ada implementation.

DEC-5 Explicitly Specify the Precision Required: Each floating point or fixed point type should explicitly specify the precision, using the **delta** or **digits** accuracy definition. This will make clear any assumptions made about accuracy of calculations.

DEC-7 Use Attributes Instead of Explicit Constraints: Consider the following example from [NISSE84].

```
A: array (DISCRETE_TYPE) of F;
  ...
for I in DISCRETE_TYPE loop
  exit when A(I) < SUM * F'EPSILON;
  SUM := SUM + A(I);
end loop;
```

This example assumes that the series A(1) + A(2) + A(3) + ... converges when all terms are positive. Because the loop depends on F's model numbers and not on explicit constraints, all Ada implementations should have the same accuracy.

DEC-8 Declare invariants as constants

In general, objects should be declared as constants if they are invariant. Declare pi as:

```
p: constant float :=3.1416;
```

Do not declare pi as:

```
p: float :=3.1416;
```

Eliminated, is the possibility of unintentional change of the invariant value.

For example, declare:

```
stack_index: stack_index_type := 0; instead of
```

Do not declare:

```
stack_index: stack_index_type;
```

Initialization precludes the use of uninitialized variables. Exceptions to the rule are limited private type variables which cannot be initialized, and situations in which performance reasons preclude initialization. In the performance category, use comments to explain where the variable is set.

Guidelines for Arrays

ARR-1 Explicitly Declare a Type to Use in Defining Discrete Ranges: Use explicitly declared types for discrete ranges. That is, use

```
type DISCRETE_RANGE is range 1..TABLE_SIZE;
type TABLE is array (DISCRETE_RANGE) of ELEMENT_TYPE;
```

instead of

```
type TABLE is array (1..TABLE_SIZE) of ELEMENT_TYPE;
```

PAPPA85, p. 28

This provides several benefits. There will be fewer logic errors when components are tailored, because the compiler will have already caught them when it checked for type inconsistencies. Also, the code will be more portable, since the compiler can select the best internal representation for the numeric type requested by the range declaration.

Unfortunately, using explicitly declared types for integer discrete ranges does not always lead to easy to read code. Type conversions may be needed to convert among explicitly declared types. The combination of long type names and required type conversions results in long multi-line Ada statements that are hard to read. Nevertheless, we believe the advantages of using explicitly declared types for integer discrete ranges outweigh the disadvantages.

ARR-2 Do not hard code array index designations: Do not hard code array index designations, as below.

```
type TABLE is array (1..50) of ELEMENT_TYPE;
```

Use types or subtypes instead, because the additional declaration will make the code more self-documenting and thus more tailorable. The upper or lower bound may be an index that will change at some time. The subtype or type declaration will allow the change to be made once instead of many times throughout the program.

Names

NAM-1 Use Descriptive Identifier Names: Use descriptive identifier names to promote readability and self-documentation. Descriptive identifier names make the code clearer. Names should be as long as necessary to provide the needed information and to promote readability. They should be considered part of the documentation of the component.

NAM-2 Keep Identifier Names Less Than 80 Characters Long: Keep identifiers less than 80 characters long, because some Ada implementations use 80 characters as the maximum identifier length. Furthermore, some display devices are limited to 80 characters, since they lack the ability to format larger strings.

NAM-3 Do Not Overload Names from Package STANDARD: Ada names predefined in **package STANDARD** should not be redefined or 'overloaded'. [RYMER86], p. 5 This keeps the reader from confusing the overloaded names with the names predefined in package STANDARD. There is an exception to this rule -- it is permissible to overload the names of operators.

Naming Conventions: Besides the above guidelines, there is no specific naming convention for identifiers in these guidelines. It is assumed that a component retrieval system will exist which will provide tools to analyze component information. The tools will have powerful analytical capabilities, so that a naming convention will not improve the analysis. It will only place unnecessary constraints on the programmer. However, if the component retrieval tools are not as powerful as anticipated, a naming convention may prove useful, and one should be considered.

Statements

Loop Statement

STA-1 Use Explicitly Declared Types for Integer Ranges in the Loop Statement*:* This will improve portability. If no type name is specified, INTEGER is used as the default, which can result in a discrete range being invalid under some Ada implementations. By using type designations, the logic can be more independent of the data.

The following example shows a loop range that should not be used.

```
for I in 1..MAX_NUM_APPLES ...
   ...
end loop;
```

Instead, do the following.

```
type APPLE_COUNT_TYPE is range 1..MAX_NUM_APPLES;
for I in APPLE_COUNT_TYPE loop ...
   ...
end loop;
```

STA-2 Exit Basic LOOPs only via Conditional
Exit and Return Statements.

Implicitly, this eliminates unconditional returns, planned Exceptions, and Goto statements as methods for exiting basic Loops.

For example, use:

```
   clear_stack:
   loop
 begin
   pop;
  exception
  when stack_empty_condition =>
   exit clear_stack;
  end;
 end loop clear_stack;
```

Do not use:

```
   clear_stack;
   loop
    exit clear_stack when stack_is_empty;
    pop;
   end loop clear_stack;
```

Exceptions used to alter the flow of control in non-error conditions, inhibit maintainability and thus portability. Gotos should be reserved for atypical situations because they inhibit maintainability. An Unconditional exit from a basic Loop implies the basic Loop structure is meaningless.

If Statement

STA-3 Use Elsif for Nested If Statements: This reduces the nesting levels of the **if** statements, giving the code a clean, uncluttered appearance. It also emphasizes the equal status of each **if** statement.

The following is an example from BARNE84, p. 50 of nested **if** statements.

```
if ORDER = LEFT then
   TURN_LEFT;
else
   if ORDER = RIGHT then
      TURN_RIGHT;
   else
     if ORDER = BACK then
        TURN_BACK;
     end if;
   end if;
end if;
```

The example below shows the above example with <hp2>elsif</hp2>s.

```
if ORDER = LEFT then
   TURN_LEFT;
elsif ORDER = RIGHT then
   TURN_RIGHT;
elsif ORDER = BACK then
   TURN_BACK;
end if;
```

Case Statement

STA-4 Avoid Using the When Others Clause as a Shorthand Notation: The **when others** clause of the case statement should not be used as a shorthand to handle all cases that have not been listed. Instead, explicitly handle each case and omit the **when others** clause. If the component is later modified to add more values to the data type, this will call attention to the fact that the new values are not handled in the **case** statement. If the **when others** clause was used, the new data values would be handled by this clause and the operation on the data might be incorrect.

If there is a long list of conditions to be enumerated, use ranges and vertical bars to simplify listing all possible values as in the following example:

```
begin
   case X is
      when AA =>
         -- Some stuff
      when DD =>
         -- Other stuff
      when BB..CC | EE..ZZ =>
         -- The Other other stuff
   end case;
end;
```

Subunits

General Guidelines

SUB-1 Use Named Constants for Parameter Defaults: Use named constants as parameter defaults whenever they would help the reader to better understand the code. For example, this

```
procedure READ (VALUE :    out ELEMENT_TYPE;
                GROUP : in     TAG_GROUP_TYPE := DEFAULT_GROUP);
```

is easier to understand than this.

```
procedure READ (VALUE :    out ELEMENT_TYPE;
                GROUP : in     TAG_GROUP_TYPE := 0);
```

SUB-2 Named Parameters: We do not believe that a set of guidelines should require named parameter association. This should be a user-selectable option with an intelligent formatter. Until such formatters are available, the following are some recommendations on the subject.

1. If there is more than one parameter in the called subprogram, then use named parameter association. This will make the interface clear to the user and make the code self-documenting, particularly when the component user is not supplying all of the possible parameters.
2. If the called subprogram only has one parameter, then use of named parameters is up to the coder. The determining factor should be whether use of the named parameter association will improve readability. Using parameter names in the interface of single parameter function calls particularly hinders readability.

SUB-3 Make Components Complete: Reusable components should be as complete as practical, meaning that the component ideally has all operations to manipulate the given object. For example, a stack package should have such operations as PUSH, POP, CLEAR_STACK and IS_EMPTY. This insures that any stack operation needed in the future will already exist and not need to be coded.

Admittedly, this guideline cannot be fully realized in practice. Yet the goal of completeness is still useful as something to strive for.

The guideline is easier to follow if standard interfaces have been established. For example, there is a standard interface for stack packages, then it will be trivial to inspect a particular stack package to tell whether it provides all the required operations.

SUB-4 Write Each Module So It Has High Cohesion: Cohesion is a measure of the degree to which the statements in a component form a whole. The most coherent components do just one thing, whether it be manipulating an object or performing a function. Cohesion should be maximized whenever possible.

One way to achieve high cohesion is to use an object-oriented design. Such a strategy makes it easy to detect low cohesion. STDEN86

SUB-5 Use Information Hiding: There are three guidelines here.

- Only place in the specification section those declarations that must be seen externally.
- Only **with** those compilation units that are really needed. Only if the specification needs such visibility should the context clause appear in the specification; otherwise it should appear in the body. A tool could be written to catch unneeded **with**s.
- Use private and limited private types to promote information hiding.

The rationale comes from the good software engineering practice of minimizing the amount of information visible to the outside world.

Guidelines on Subprograms

SUB-6 Use Descriptive Named Constants as Return Values: Named constants should be returned whenever they would help the reader to understand the code. For example, it is more informative to return the named constant NOT_FOUND than to return the value -1.

Exception Handling

As stated earlier in this document, good exception handling is important to software reuse for several reasons.

- Components with good error / exception handling have safety built in.
- Errors are isolated and well-documented.
- The way interfaces work is made clear. There are fewer hidden assumptions.
- The users have the freedom to decide whether to propagate exceptions further, to retry the operation that raised the exception, to abandon the operation, or to continue regardless.
- Good exception handling makes components more tailorable and thus more reusable.

Exception Handling Design

EXP-1 Avoid the When Others Construct with the Null Statement: Use of the **null** statement suggests that the exception is not used for an abnormal condition.

```
begin
  loop
    ...
    raise MISCELLANEOUS_ERROR;
    ...
  end;
exception
 when others =>
  null;
 end;
 -- rest of normal program code
```

In the above example a **raise** statement is used to exit the loop and to continue executing normal control flow. This implies that there never was an abnormal condition.

EXP-2 Avoid pragma SUPPRESS: The Ada Language Reference Manual ADA83 does not require that **pragma SUPPRESS** be implemented. **Pragma SUPPRESS** does not guarantee that exceptions will not be propagated to a unit for which exception suppression is in effect. The execution of a program is erroneous if an exception occurs while **pragma SUPPRESS** is in effect.

EXP-3 Handle Exceptions Close to Where They Are First Raised: This gives the exception handler access to local data, which can be used to respond to the exception. It also avoids losing visibility to the exception name.

When an exception is propagated to a scope outside its visibility, its name is lost. The exception can only be handled by a **when others** handler. Such exceptions might be unwittingly handled by a handler that was never intended to handle the exception.

Below is an example of an exception handler making use of local data.

```
package body SEQUENTIAL_ACCESS_METHOD is

   CURRENT_RECORD : RECORD_IDENTIFIER := START_OF_FILE;

   procedure GET (FILE : in FILE_TYPE;
                  REC : out RECORD_TYPE) is
      VALUE : RECORD_TYPE;
   begin
      ...
      VSAM.VGET(CATALOG => FILE,
                INDEX   => CURRENT_RECORD + 1,
                DATA    => VALUE);
      ...
      exception
        when DEVICE_ERROR =>
          ERROR_IO.LOG("Error Occurred Reading Record" amp
                RECORD_IDENTIFIER'IMAGE(CURRENT_RECORD + 1));
   end GET;
end SEQUENTIAL_ACCESS_METHOD;
```

EXP-4 Imbed Potential Elaboration Exceptions in a Frame: Unless special provisions are made, elaboration exceptions are not handled in the unit being elaborated. Consider the following example.

```
package ELABORATION_EXCEPTION_PKG is
  S : STRING (1..2) := &amp;quot;Causes Constraint_Error&amp;quot;;
end ELABORATION_EXCEPTION_PKG;
```

The Constraint_Error exception that is generated is not handled inside the package; it is propagated out.

The solution is to imbed potential elaboration exceptions in a frame. To do this, do the following.

- Move declarations to declare blocks inside executable regions,
- Do initializations inside executable regions, and
- Encapsulate initializations within a subprogram to take advantage of Ada's strong typing, as in the following example.

```
with INITIALIZATION_PKG;
package ELABORATION_EXCEPTION_PKG is
  S : INITIALIZATION_PKG.NAME_TYPE := INITIALIZATION_PKG.SET_NAME;
end ELABORATION_EXCEPTION_PKG;

package INITIALIZATION_PKG is
   subtype NAME_TYPE is STRING(1..2);
   function SET_NAME return NAME_TYPE;
     -- guarantees no CONSTRAINT_ERROR
end INITIALIZATION_PKG;
```

Exception Propagation

EXP-5 Do Not Propagate an Exception Where Its Name Is Not Visible: Do not propagate an exception beyond where its name is visible. Otherwise, it can only be handled by a **when others** handler.

EXP-6 Do Not Propagate Predefined Exceptions Without Renaming Them: Predefined exceptions have no corresponding **raise** statement in the source code, so it is not always obvious that an exception can be propagated. Predefined exceptions can be raised by many operations, making them difficult to locate. Renaming predefined expressions makes it easier to pinpoint the exact cause of each exception. For example, the predefined exception STORAGE_ERROR might be propagated as MEMORY_FULL.

Use of Exception Handling

EXP-7 Do Not Execute Normal Control Statements from an Exception Handler: Only use exception handling for abnormal control flow, not for normal control.

Below is an example of poor use of exception handling.

```
begin
  loop
    TEXT_IO.GET(DATA_FILE,DATA_VALUE);
    ...
  end loop;
exception
  when TEXT_IO.END_ERROR(DATA_FILE) =>
      -- execute the rest of the program here
end;
```

In contrast, the following shows equivalent code without the use of an exception handler.

```
while not TEXT_IO.END_OF_FILE(DATA_FILE) loop
   TEXT_IO.GET(DATA_FILE,DATA_VALUE);
   ...
end loop;
-- execute the rest of the program here
```

Exception Documentation

EXP-8 Document All Exceptions Which Will Be Propagated from an Operation in an Exception Documentation Block: An Exception Documentation Block shows which operations raise which exceptions under what conditions. In this block, describe all the conditions that cause each exception to be raised, including predefined exceptions. This will help other developers in making their designs complete.

Be sure to clearly associate each exception with every operation where the exception can be raised. If the same operation can raise an exception for different reasons, record each reason separately.

The following is an example of an Exception Documentation Block.

```
STRING_TOO_LONG : exception;
    ------------------------------------------------------------
    --
    --   Raised By              On Condition
    --   ---------              ------------
    --
    --   INSERT                 the size of the string with the
    --                          insertion exceeds MAX_DYNAMIC_
    --                          STRING_LENGTH
    --
    --   REPLACE                the size of the string with the
    --                          replaced part exceeds MAX_DYNAMIC_
    --                          STRING_LENGTH
    ------------------------------------------------------------
```

EXP-9 List all Conditions That Raise Exceptions in Subprogram Specification Comment Blocks: This includes the conditions that will cause predefined exceptions to be raised and includes multiple conditions that can cause the same exception.

Exception Handling Parameter Usage

EXP-10 Ensure That Out Parameters Cannot Be Undefined Upon Return from a Subprogram If an Exception Occurs: Never depend on the value of **out** parameters or **return** values when designing a handler response. When an exception occurs while evaluating the right side of an expression, then the current value of the variable stays the same. The values of scalar **out** parameters which are not updated are undefined. Thus, the exception handler should set the values of scalar parameters before returning.

Implementation Dependencies

Design Considerations

IMP-1 Isolate Compiler, Operating System and Machine Dependencies: To make components portable, avoid optional language features and Ada implementation dependencies. Where this cannot be done, isolate such uses, so users can plug in new versions easily. Document all such uses. Both encapsulation and documentation will reduce the effort to port a component to a new implementation.

Write code to ignore details of underlying implementations. Components should be designed without reference to the surrounding environment. Contact between a component and its environment should occur through explicit parameters and explicitly invoked subprograms. PAPPA85, p. 7

IMP-2 Avoid Optional Language Features: For example, avoid using UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION. These two procedures are optional and implementation dependent. If you use these procedures, document their use. *Environment_Imposed_Restrictions:/hp1. and Compiler_Dependent_Restrictions.*

Pragmas

IMP-3 Avoid Using Pragmas: Pragmas are instructions to a specific compiler. Pragmas generally imply environmental depenendencies and, therefore, are a negative portability indicator. For example, Pragma INTERFACE implies the existence and dependence of a component on non-ADA code. Pragma ELABORATE is needed to force elaboration order for correct compilation. Elaboration order is compiler dependent and may not be preserved with another compiler.

Pragma INTERFACE may be needed to specify interfaces with subprograms of other languages. Pragma ELABORATE may be needed to insure that a program is correctly elaborated no matter what compiler is used, since elaboration order varies from compiler to compiler. However, take care that pragma ELABORATE is essential and not needed because the component is overly complex.

IMP-4 If Pragmas Are Used, Isolate and Thoroughly Document Them: If they must be used, they should be isolated as much as possible.

Those components which use pragmas should be documented, pointing them out and describing their effects. Pragmas.

For example, use of **pragma INLINE** *in a reusable component should be documented. Its use can force the user's code to depend on the body of the reusable component. Since this effect is usually unexpected, take care to insure that the reuser is aware of the compilation issues caused by it.*

Input and Output

General Guidelines

IO-1 Encapsulate I/O Uses Into a Separate I/O Package: All input/output utilities should be isolated into I/O packages. This will make it easy for users to adapt the component to different machines and operating systems.

Guidelines on Specific I/O Procedures and Functions

IO-2 Do Not Rely on NEW_PAGE: The Ada language standard does not specify the value of a page terminator. Thus, the control characters may be non-portable across printers. One solution is to always direct output to a file, which can then be filtered and altered to suit the device the output is ultimately destined for.

IO-3 Document Implementation Dependent Procedures: Use of the following procedures could result in portability problems. Dependencies on such procedures should be documented. Portability_Restrictions.

- *COL -- Depends on the implementation-defined subtype POSITIVE_COUNT.*
- *DIRECT_IO.READ -- Reads from an index whose range POSITIVE_COUNT is implementation defined. The DIRECT_IO.WRITE procedure could also cause similar problems.*
- *ENUMERATION_IO.GET -- Returns an* **out** *parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input. It also has a WIDTH parameter of the implementation-defined type FIELD.*
- *FIXED_IO.GET -- Returns an* **out** *parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input. It also has a WIDTH parameter of the implementation-defined type FIELD.*
- *FIXED_IO.PUT -- Has a WIDTH parameter of the implementation-defined type FIELD.*
- *FLOAT_IO.GET -- Returns an* **out** *parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input. It also has a WIDTH parameter of the implementation-defined type FIELD.*
- *FLOAT_IO.PUT -- Has a WIDTH parameter of the implementation-defined type FIELD.*

- *GET_LINE -- Returns an* **out** *parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input.*
- *INDEX -- Uses an index whose range POSITIVE_COUNT is implementation-defined.*
- *INTEGER_IO.GET -- Returns an* **out** *parameter of the predefined types POSITIVE or NATURAL to specify the LAST character input. It also has a WIDTH parameter of the implementation-defined type FIELD.*
- *INTEGER_IO.PUT -- Has a WIDTH parameter of the implementation-defined type FIELD.*
- *LINE -- Depends on the implementation-defined subtype POSITIVE_COUNT.*
- *LINE_LENGTH -- Depends on the implementation-defined type COUNT, whose upper bound varies with each implementation.*
- *PAGE -- Depends on the implementation-defined subtype POSITIVE_COUNT.*
- *PAGE_LENGTH -- Depends on the implementation-defined type COUNT, whose upper bound varies with each implementation.*
- *SET_COL -- Depends on the implementation-defined subtype POSITIVE_COUNT.*
- *SET_LINE -- Depends on the implementation-defined subtype POSITIVE_COUNT.*
- *SET_INDEX -- Uses an index whose range POSITIVE_COUNT is implementation-defined.*
- *SET_LINE_LENGTH -- Depends on the implementation-defined type COUNT, whose upper bound varies with each implementation.*
- *SET_PAGE_LENGTH -- Depends on the implementation-defined type COUNT, whose upper bound varies with each implementation.*
- *SIZE -- Uses an index whose range POSITIVE_COUNT is implementation-defined.*

[*MATTH87b, p. 2*]

Furthermore, using procedures SKIP_LINE and NEW_LINE to skip more than one line at a time may lead to portability problems, since they depend on the implementation-defined subtype POSITIVE_COUNT. Skipping one line will not cause any problems; however, skipping multiple lines may not be portable depending on the constraint set by the Ada implementation. [*MATTH87b, p. 2*]

File Handling

IO-4 Close Files Before a Program Completes: Different Ada implementations handle unclosed files in different ways. The state of unclosed files after program termination is undefined. To increase the reusability of a component, close all files before a subprogram terminates normally or abnormally. Be sure to verify that a file is open before closing it so that the exception STATUS_ERROR is not raised.

I/O of Access Types

IO-5 Do Not Input or Output Access Types: The effect of I/O of access types is undefined. If used, it may lead to components that are not portable. To output an object pointed to, output the object. To output the address of an object pointed to, output the address of the object using SYSTEM.ADDRESS. [*MATTH87b, p. 1*] *Document the use of SYSTEM.ADDRESS.*

3. References

There are numerous coding guidelines available, particularly for Ada. The following is a list of references for the STARS reusability coding guidelines.

[ADA83] *Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, February, 17 1983.*

[AHO74] *Aho, A. V., J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Readding, Mass.: Addison-Wesley, 1974.*

[AUSNI85] *Ausnit, Christine, Christine Braun, Sterling Eanes, John Goodenough, Richard Simpson, Ada Reusability Guidelines, SofTech, Inc., April 1985.*

[BARNE84] *Barnes, J.G.P., Programming in Ada, 2nd edition. Addison-Wesley Publishers Limited, 1984.*

[BENTL85] *Bentley, Jon, "Programming Pearls," Communications of the ACM, vol. 28, no. 7 July 1985.*

[BOOCH87] *Booch, Grady, Software Components With Ada. The Benjamin/Cummings Publishing Company, Inc., 1987.*

[EMBLE87] *Embley, David W. and Woodfield, Scott N., "Cohesion and Coupling for Abstract Data Types." Proceedings, Sixth Phoenix Conference on Computers and Communications, Phoenix, Arizona, February 1987.*

[EVBSE87] *EVB Software Engineering, Inc., Creating Reusable Ada Software, 1987.*

[IBM0340] *IBM Systems Integration Division, Informal Technical Report on Findings During the Rebuild of Common Capabilities, CDRL Sequence No. 0340, February 19, 1989.*

[IBM0360] *IBM Systems Integration Division, Reusability Guidelines, CDRL Sequence No. 0360, December 17, 1988.*

[IBM0370] *IBM Systems Integration Division, Reusable Component Data Analysis, CDRL Sequence No. 0370, February 10, 1989.*

[IBM0380] *IBM Systems Integration Division, Consolidated Reusability Guidelines, CDRL Sequence No. 0380, March 21, 1989.*

[IBM0460] *IBM Systems Integration Division, Repository Guidelines and Standards, CDRL Sequence No. 0460, March 17, 1989.*

[IBM0520] *IBM Systems Integration Division, Long Term Configuration Management Plan for the STARS Repository, CDRL Sequence No. 0520, March 17, 1989.*

[IBM0710] *IBM Systems Integration Division, DTD Definition: Internal Documentation, CDRL Sequence No. 0710, January 16, 1989.*

[MATSU84] *Matsumoto, Y., " Experiences in Promoting Reusable Software Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, vol. SE-10 (5), September 1984.*

[MATTH87a] *Matthews, E. R., IBM Federal Systems Division Guide for Reusable Ada Components (Draft), September 17, 1987.*

[MATTH87b] *Matthews, E. R., "Observations on the Portability of Ada I/O," ACM SIGAda Letters, vol. VII, no. 5, September/October 1987.*

[MCILR68] *McIlroy, M. D., "Mass Produced Software Components," Report on a conference by the NATO Science Committee, Garmisch, Germany, October 7-11, 1968.*

[MENDA88] *Mendal, Geoffrey O., "Three Reasons to Avoid the Use Clause," ACM SIGAda Letters, vol. VIII, no. 1, January/February 1988.*

[NISSE84] *Nissen, John and Peter Wallis, Portability and Style in Ada, Cambridge University Press, 1984.*

[PAPPA85] *Pappas, Frank, Ada Portability Guidelines, SofTech, Inc., March 1985.*

]RACIN88] *Racine, Roger, "Why the Use Clause is Beneficial," ACM SIGAda Letters, vol. VIII, no. 3, May/June 1988).*

]ROSEN87] *Rosen, J. P., "In defense of the 'use' clause," ACM SIGAda Letters, vol. VII, no. 7, November/December 1987.*

]RYMER86] *Rymer, John and McKeever, Tom., The FSD Ada Style Guide, 1986.*

[SOMM89] *Sommerville, I., Software Engineering, 3rd. edition, Addison-Wesley, 1989.*

]STDEN86] *St. Dennis, R., P. Stachour, E. Frankowski, and E. Onuegbe, "Measurable Characteristics of Reusable Ada Software," ACM SIGAda Ada Letters, vol. VI, no. 2, March/April 1986.*

]STEVE74] *Stevens, W. P., G. J. Myers, and L. L. Constantine, "Structured design." IBM Systems Journal, 1974, no. 2.*

]UNISYS0340] *Unisys Corporation, Draft Technical Report on Reusability Guidelines, CDRL 0340, February 14, 1989.*

4. Glossary

The following terms and definitions describe component attributes and design issues, as used in these guidelines.

anonymous type *A type without a simple name.*

cohesion *A measure of the degree to which the code in a module forms a coherent whole.*

contact *The contact is the person in the producing company who is the 'point of contact' for that particular part or product. Point of contact refers to the person who is familiar with the product, and can either answer questions about it, or can refer people to someone who can answer them.*

coupling *A measure of how much components or modules depend on each other. Coupling depends on the interfaces between modules, the data that pass between them, and the control relationships.*

dynamic stack *The stack of calls made at runtime.*

exception documentation block *A comment that documents an exception.*

frame *An Ada language construct that surrounds an exception handler. A frame can be a block statement or the body of a subprogram, a package, a task, or a generic.*

functional completeness *The idea that components should have all functions and operations required for current and future needs.*

independence *The ability of a component to be used with different compilers, operating systems, machines and applications than those for which it was originally developed. Independence is closely related to portability.*

maintainability *The ease of modifying a component, whether it be to meet particular needs or to fix bugs.*

order (of an algorithm) *A measure of the computational efficiency of an algorithm, expressed in terms of the frequency of some key operation. For more information, see A1IO74.*

overloading *The property whereby Ada literals, operators, identifiers, and aggregates can have unambiguous alternative meanings.*

platform *Platform refers to the architecture for the system for which the product is intended (hardware, operating system, and Ada compiler) Some products may be intended for several different platforms. Platforms listed should also indicate whether they are host platforms, target platforms, or both.*

portability *The ability of an application or component to be used again in a different target environment than the one it was originally built for. The phrase target environment may be defined broadly to include operating systems, machines, and applications. To be ported effectively, components may need to be tailored to the requirements of the new target environment. See also reusability and independence.*

reliability *The extent to which a component performs as specified. A reusable component performs consistently with repeated use and across environments (that is, operating systems and hardware).*

reusability *The ability to reuse a software component or to use it repeatedly in applications other than the one for which it was originally built. In order to be effectively reused, the component may have to be tailored to the requirements of the new application. See also portability.*

subprogram specification comment block *A comment block that accompanies a subprogram specification, giving its name and a description of its function.*

tailorability *The ease of modifying a component to meet particular needs. It should be distinguished from maintainability, which includes tailorability, but also includes the idea of corrective maintenance (fixing bugs).*

Index